XML2CSV-GENERIC-CONVERTER

VERSION

1.0.0

"EQUINOX"

AUTHOR

LAURENT POPIEUL

lochrann@rocketmail.com

Last update : Saturday, May 3, 2014

Contents

1 - Background

Date	Author	Version	Description
03/21/2014	Laurent Popieul	1.0.0 ("Equinox")	From scratch.
04/21/2014	Laurent Popieul	1.0.0 ("Equinox")	Adjustments.

Written with Open Office Writer 4.0.1 using Cédric Billard's documentation model available at http://templates.openoffice.org/fr/template/documentation.

2 - Audience

XML2CSVGenericConverter user documentation - Copyright 2014 Laurent Popieul.

This document describes what the program does, how to execute it, and its runtime parameters.

The targeted audience consists of programmers and advanced end users.

XML2CSVGenericConverter and this accompanying documentation come with ABSOLUTELY NO WARRANTY. This is free software, and you are welcome to redistribute it under certain conditions.

3 - XML2CSV Generic Converter

3.1 - XML2CSV: *who cares?*

XML has become a standard for data exchange between heterogeneous systems but it's quite a verbose language and some files grow beyond reason.

Fortunately XML files can be zipped easily for transport and computer experts know how to build efficient programs which can deal with XML monsters.

From time to time however, end-users need to pick up data from XML files by hand and check them.

For instance:

- because data were expected in the last XML files but can't be found anywhere in the IS, and though the IT keeps telling us everything is OK and no error occurred, someone has to check things out explicitly;
- because the program displayed funky figures and the calculation has to be double-checked by hand from the XML source.

Small XML files are no problem at all, and average sized XML files can be imported by a spreadsheet and easily reworked from it.

Trouble comes with big XML files and it's not uncommon to see products fail, freeze, crash or collapse when confronted to XML of a respectable size.

There are two approaches in such situations:

1. to have the file opened by a smart editor (such as Notepad++) and then filter/clean the data in order to discard everything but the meaningful tags (for instance with carefully devised *regular*

expressions);

2. or to rely on a utility program which will extract data from this nasty hotchpotch.

I wrote this XML2CSV converter because I was not too pleased with option 1. 's error prone manipulations, and not exactly satisfied with what I had up my sleeve so far to handle option 2.

I wanted something generic which would basically convert all XML data to CSV for later opening with a spreadsheet without bugging me with things like enclosing blocs or repeated elements.

XML is simple, CSV is simple, so why would XML ⇨ CSV have to be complicated?

I wanted some extra options too, making it possible either to select some XML tags and have the program extract them only, or to select some XML tags and have this time the program extract everything but those ones.

I must admit that the resulting program was bigger in the end than what I expected in the first place. Indeed, placing cells at the right place can be a tricky job sometimes, especially when a new XML tag might pop up anytime because you have no XML schema telling you what is possible and what is not.

Anyway, I was personally satisfied enough with the result – at least for my own needs – and decided to share it with other fellow XML consumers who might look for something alike.

3.2 - XML2CSV: *how to?*

The generic converter is packaged as an executable Java Jar file.

The core Java class is named `XML2CSVGenericGenerator` and holds a `generate` method in charge with both XML ⇨ CSV conversion and CSV output file generation.

A wrapping class named `XML2CSVConsoleCommand` is responsible for interactions with the end user and, just like its name suggests it, in charge of console command execution.

This wrapping class is declared in the Jar file as the executable class of the package which means that when the Jar is executed `XML2CSVConsoleCommand` is run automatically.

In other words, showing you how to run XML2CSV conversion goes back to show you how to execute a Java Jar with or without parameters. In our case, it amounts to:

- copy `XML2CSVGenericConverter_V1.0.0.jar` to a local directory on your machine;
- open a console window with Java runtime resources available – 1.6 or higher ([1]);
- go to the directory where you copied `XML2CSVGenericConverter_V1.0.0.jar`;
- type `java -jar XML2CSVGenericConverter_V1.0.0.jar` and press "enter" to execute the JAR without parameter (you will get a WARNING/ERROR message telling you that parameter `-i` is missing ([2]), and another one inviting you to rerun the program with option `-h` in order to display the on-line help);
- type `java -jar XML2CSVGenericConverter_V1.0.0.jar -h` and press "enter" to execute the JAR with parameter `-h`, in order to display the on-line help;
- type `java -jar XML2CSVGenericConverter_V1.0.0.jar` followed a list of parameters (complete list provided at § 4 -) and press "enter" to execute the JAR with the options you chose.

If you are a more advanced Java user then you might prefer to execute the main JAR main class yourself with the following command:

1 *Type:* `java -version` *and press "Enter". If the system is well configured you should see a message displaying the current Java version on your system. If you see an error message then you should make Java available on your system first before you try and perform XML2CSV conversion again.*

2 *The only mandatory parameter.*

`java utils.xml.xml2csv.XML2CSVConsoleCommand` {parameters}

where {parameters} represents your runtime parameters in sequence (see § 4 -).

Don't forget to add `XML2CSVGenericConverter_V1.0.0.jar` to your local Java `CLASSPATH` before.

3.3 - XML2CSV: *what for?*

3.3.1 - A few definitions, to settle things

In an XML structure a tag might contain other tags or data.

In order to settle things, I will call in the rest of this document a tag containing plain data a *leaf* tag, and a tag containing other tags a *composite* tag, *parent* of the *children* tags enclosed in it.

A *composite* tag can contain *composite* tags, *leaf* tags, or a mix.

Composite and *leaf* tags can be *repeated* in a valid XML file, or remain *single*. A *repeated* tag will appear at least twice within the same *parent* tag.

A tag which does not appear systematically at its position in a *repeated composite* tag is named an *optional* tag.

In the sample xml file below `<Row>` is a *composite repeated* tag as well as the `<Data>` tag within a `<Row>`.

sample.xml

The `<Other>` tag inside a `<Row>` is a *composite* tag too but it is not *repeated* in each `<Row>`.

The `<Data0>`, `<Amount>` and `<Data1>` tags are *single leaf* tags inside the *composite* `<Data>` tag and the three of them are *optional* tags because they do not always exist in each `<Data>` tag ([3]).

The `<Date>` tag inside a `<Row>` is a *repeated optional leaf* tag.

3.3.2 - What does XML to CSV conversion exactly do?

The XML ⇨ CSV converter generates one column for each XML *leaf* tag.

The content of each *leaf* tag is put in a row in the corresponding column, one under the other if it is a *repeated leaf* tag.

Data which are *directly* or *indirectly related* might appear on the same line for a more compact representation, or left in separate rows.

Here is my definition of *related* elements:

- if you consider one initial XML *single leaf* tag and if you follow its *parents* back to a certain enclosing level in the XML file then each *single leaf* tag which is also a *child* of one of those chained *parents* is *indirectly related* to the initial XML *leaf* tag, provided that all the *parents* of the chain be *single* too (apart from the last enclosing *parent* which might be *repeated* or not);
- if you consider one initial *single leaf* XML tag, all the other XML *single leaf* tags which have the same *parent* tag are *directly related* to that initial tag (no matter if the *parent* is *repeated* or not).

If you look at the `<Data0>` tag inside the `<Data>` block of the 1st `<Row>` in the previous file `sample.xml` you

3 *In fact, all leaf tags in the sample xml file are optional.*

will see that it is closely related to the `<Amount>` and the `<AddAmount>` elements of that 1st `<Row>`, so `true`, `123` and `12` might appear on the same CSV output line.

Grouping *directly and indirectly related* tags on the same line can be interpreted as linking them to the same *adoptive parent*.

Once *related* tags have been grouped on the same line they might be copied back in lines associated with their *half-brothers* (that is, *repeated leaf* tags which have the same *parent* as the *adoptive parent* of these *related* tags) or even more remote *half-distant-nephews* (*repeated leaf* tags which have the *adoptive parent* as *ancestor*).

Directly related fields may be copied back to their *distant-nephews* too (that is: *single leaf* tags which have the *parent* of these *related* tags as *ancestor*).

You might also safely consider copying *related* tags back to *single* or *repeated leaf* tags which have the *adoptive parent* as *ancestor* provided that at least one of these chained *parents* be *repeated*.

In the previous file `sample.xml`, `true`, `123` and `12` might be duplicated back on the lines displaying `ee1` and `ee2`, and on the lines displaying `SD1a` and `SD1a` as well.

The CSV output file could look like the `sampleOutput.csv` file below, but it could be slightly different too, depending on your chosen packing/duplicating tactic.

sampleOutput.csv

However, whatever the tactic, *repeated* tags are never *related* with one another and, as such, can never be mixed and must always appear on separate lines:

- one line under the other for the same *repeated* tag;
- in different line blocs for distinct *repeated* tags (no matter if they have the same *parent* or not).

`XML2CSVGenericConverter` offers so far four different packing/duplicating strategies, one for each optimization flavor available at runtime. They are described in the next paragraphs.

3.3.3 - Optimization levels

3.3.3.1 - Raw level

Raw optimization level stands for no optimization at all, and no data packing at all.

In this mode the content of each *leaf* element is put alone on a new CSV line at its expected column index, and the output CSV file will hold as many lines as there were XML *leaf* elements.

Because no data buffering of any kind is needed for such an elementary behavior, the data & size of the XML input file(s) have absolutely no impact over performance even in nasty scenarios such as the one implying a heavily *repeated leaf* tag (say: `<paininthe>BACK</paininthe>` repeated thousands of time in a row and you'll get it).

The file below is the raw XML ⇨ CSV conversion of the previous `sample.xml` file generated by `XML2CSVGenericConverter`:

sampleRawOutput.
csv

3.3.3.2 - Standard optimization

Standard optimization consists in packing *directly related* fields on the same CSV line at each level of the tag hierarchy.

This mode produces fairly compact CSV output files in as much as:

- each XML element content appears once only in the output file(s);
- packing *directly related* fields reduces the number of lines.

Data layout in the CSV file(s) is also very much like what it is in the original XML counterpart.

For all these reasons, I chose it as the default optimization triggered when no particular parameter (`-r` or `-x`) is provided on the command line.

The file below is the standard XML ⇨ CSV conversion of the previous `sample.xml` file generated by `XML2CSVGenericConverter`:

sampleStandardOut
put.csv

Field packing is performed in memory and requires that a consistent amount of data be buffered, that is, to wait until the last XML closing tag of the last *parent* of *leaf* tags be read.

This approach makes it possible to keep the buffer fairly small because it guarantees that, in practice, it won't contain more than one XML repeated block in memory (with the previous `sample.xml` file: more than one `<Row>` block).

Each time a buffer reaches consistency, it is packed, then flushed to the CSV output, and buffering starts fresh anew.

It works pretty well with real world XML files, big or small, but advanced users should be aware that some exotic XML flavors would still lead to memory overload, namely something like:

- an XML file bearing a flock (say: tens of thousands) of different *leaf* tags inside the same row;
- an XML file bearing a heavily *repeated leaf* tag (again, tens of thousands or more).

Indeed, the buffer would grow accordingly to fit the data before the program actually gets a chance to flush it (waiting to reach consistency) at the risk of an `OutOfMemoryError` (in the 1st case: caused by a outrageous amount of buffer columns; in the 2d case: caused this time by an outrageous amount of buffer lines).

Fortunately, it is generally considered very bad practice (and I just can't imagine which dark purpose it could serve to compose such alien XML).

However, should anybody have to deal with hostile extra galactic XML, the previous raw optimization level would provide a quick fix and still produce some operable output (see § 3.3.3.1 -).

3.3.3.3 - Extensive optimization – variant 1

In its first variant, extensive optimization consists in:

- packing *directly related* fields on the same CSV line, at each level of the tag hierarchy, just like the previous optimization does;
- at each level of the tag hierarchy, copying the previous *related* fields back into lines associated with *half-brothers* (CSV lines which are associated with the content of a *repeated leaf* tag having the same *parent*) or *half-distant-nephews* (CSV lines which are associated with the content of a

repeated leaf tag having the same *ancestor*);

- replicating the *directly related* fields to their *distant-nephews* (CSV lines associated with the content of a *single leaf* tags having the same *ancestor*).

In other words extensive optimization - variant 1 - is a standard CSV optimization of *single leaf* tags plus copy back into CSV lines representing *repeated leaf* tags of the same XML level or deeper level in the same XML block, plus copy back into deeper *single leaf* linked lines in the same XML block.

The file below is the extensive #1 XML ⇨ CSV conversion of the previous `sample.xml` file generated by `XML2CSVGenericConverter`:

sampleExtensiveV1
Output.csv

This extensive optimization uses the same buffering strategy as the one used for standard optimization (see § 3.3.3.2 -).

Extensive optimization #1 may not produce the best XML ⇨ CSV conversion results (that is: the less CSV lines possible) because its standard packing strategy handles *directly related* fields only and casts aside *indirectly related* fields (see § 3.3.2 -).

Another extensive optimization, based this time on both *directly* and *indirectly related* fields packing, is described in the next paragraph.

3.3.3.4 - Extensive optimization – variant 2

In its second variant, extensive optimization consists in:

- packing *directly* or *indirectly related* fields on the same CSV line, at each level of the tag hierarchy, associated with the same *adoptive parent*;
- at each level of the tag hierarchy, copying the previous *related* fields back into lines associated with *half-brothers* (CSV lines which are associated with the content of a *repeated leaf* tag having the same *parent*);
- at each level of the tag hierarchy, copying those *related* fields back to *single* or *repeated leaf* tags which have the *adoptive parent* as *ancestor* provided that at least one of these chained *parents* be *repeated*.

The file below is the extensive #2 XML ⇨ CSV conversion of the previous `sample.xml` file generated by `XML2CSVGenericConverter`:

sampleExtensiveV2
Output.csv

It looks pretty much like what extensive #1 XML ⇨ CSV conversion does, which is not very surprising (and maybe quite reassuring too).

However, extensive #2 XML ⇨ CSV conversion might produce better results (that is: less CSV lines) than extensive #1 XML ⇨ CSV conversion with XML containing a lot of *single composite* tags and a lot of *single leaf* tags.

This extensive optimization uses the same buffering strategy as the one used for standard optimization (see § 3.3.3.2 -).

3.3.4 - Filter files

3.3.4.1 - Positive filter file

When left on its own, `XML2CSVGenericConverter` assumes that all XML *leaf* tags are interesting and should appear as columns in the CSV output, and behaves accordingly.

This said, the end user may select an explicit subset of XML *leaf* tags, gather them in an appropriate file listing the corresponding XML element Xpaths and rerun `XML2CSVGenericConverter` with option `-p` to have the program extract those fields only (see § 4.1 -).

This file, which I named a *positive filter file* for obvious reasons, is expected `UTF-8` encoded by the program without Byte Order Mark (BOM).

It might contain comments, that is, lines beginning with two consecutive dashes (`--`), and might contain blank lines as well.

Xpaths have to be listed one under the other (one per line), and the program will smoothly keep the same order for the generated columns in the output file(s).

Duplicates Xpaths are sorted out automatically.

Junk Xpaths which do not exist in the XML input ([4]) are discarded, and an appropriate warning message is displayed (but the program will stop if all the Xpaths listed in the *filter file* happen to be incorrect).

Xpaths of *composite* tags might be mixed with Xpaths of *leaf* tags, and the program will automatically replace a *composite* Xpath with all the *leaf* Xpaths it contains, eliminating duplicates.

For instance the *positive filter file* below, consistent with our initial `sample.xml` XML file, contains three *leaf* Xpaths (`Root.Row.Data.Data0`, `Root.Row.Data.Amount` and `Root.Row.Date`) plus one *composite* `Root.Row.Other` Xpath that the program replaces automatically with the *leaf* Xpaths it contains (namely `Root.Row.Other.Data.Data1` and `Root.Row.Other.Data.Data2`).

sampleFilterFile.txt

Applying this *filter file* in a *positive filtering* context on `sample.xml` would extract the `Data.Data0`, `Data.Amount`, `Date`, `Other.Data.Data1` and `Other.Data.Data2` tag values only and put them in a CSV output file.

The file below is the extensive #1 XML ⇨ CSV *positive* output generated by `XML2CSVGenericConverter` against `sample.xml` when plugged onto the previous `sampleFilterFile.txt` with option `-p`:

samplePositiveOutp
ut.csv

Filter files allow attribute filtering (provided that you decided to have attributes extracted by adding `-a` on the command line) which means that Xpaths listed in a regular *positive |negative filter file* might represent XML elements or attributes.

You will recognize an element's attribute from the column header name in a CSV output file. Indeed, a `@`

4 *I'd rather say: which do not exist in the in-memory structure representation of the XML input that the program builds on the fly. This remark is not as mundane as it seems, because, as you will read in § 3.3.5 - , the program just ignores data which do not comply with its inner structure representation, solely build out of one of the XML input files.*

separation character replaces the dot between the attribute name and its enclosing element Xpath (for instance: `Root.Row.Data.Amount@Currency`, if `Currency` is an attribute of the `Amount` tag).

This Xpath representation is also what the program expects in *filter files* to feature an attribute: you might put `Root.Row.Data.Amount` and `Root.Row.Data.Amount@Currency` in sequence in a *filter file* and run the program with option `-p` to extract both the corresponding amount and its currency.

I chose to display attributes by default in columns placed after the one dedicated to their enclosing element, just like spreadsheets do.

This said, because you have control over the column order in *filter files* you might choose to reverse this default order at your will (with the previous example: it you put `Root.Row.Data.Amount@Currency` before `Root.Row.Data.Amount@Currency` in your *filter file*) or even separate an element from its attributes.

Frankly, I didn't want to enclose attribute support in version `1.0.0` but finally gave it better thoughts and shipped the product with minimal support for *leaf* elements first, and now with full attribute support ([5]).

3.3.4.2 - Negative filter file

Negative filtering is just the reverse of *positive filtering* (see § 3.3.4.1 -).

A *negative filter file* has the same structure as a *positive filter file* (`UTF-8` encoded without BOM, *composite* or *leaf* Xpaths in columns, and so on) but when fed with a *negative filter* with command line option `-n` (instead of `-p`) the program extracts all XML fields but those listed in the *negative filter file*.

The file below is the extensive #1 XML ⇨ CSV *negative* output generated by `XML2CSVGenericConverter` out of `sample.xml` when plugged onto the previous `sampleFilterFile.txt` with option `-n`:

sampleNegativeOut
put.csv

3.3.5 - Structure analysis and XML processing

3.3.5.1 - Principle

`XML2CSVGenericConverter` can process one or several files in a row. Indeed, option `-i` might indifferently point to a single XML file, or to a directory ([6]) from which the program will process all XML files ([7]) in sequence.

XML input files processed consecutively are expected alike, that is, sharing the same general XML structure.

This is important because `XML2CSVGenericConverter` performs an in-memory analysis of that structure before the data of the first XML file is processed, and confidently relies later on this structure representation to fetch data.

In details, when structure analysis phase is triggered the program picks up one of the XML input files at random and parses it, not in order to retrieve its data but to look at the tag hierarchy imbrication (figuring *composite* and *leaf* tags), the tag cardinalities (figuring *single* or *repeated* tags), and the data types as well (something which has no real purpose so far, but who knows what could come up with version `1.1.0`?..).

5 *The effect of spring sunlight on a tired soul is just amazing.*

6 *Absolute or Jar-relative paths are welcomed.*

7 *That is, files with a* `.xml` *extension.*

3.3.5.2 - XML template file

I name the XML file which serves as model during the structure analysis phase the XML *template file*.

The random default *template file* picking lottery described in the previous paragraph might be bypassed anytime using the `-t` command line option, which makes it possible to provide the name (without its path) of the particular XML input file which should be used as *template* among those listed with option `-i`.

I added this option to let advanced users who have comparable files to process and who know that one particular XML input file is the most complete (I mean: with all its optional tags appearing at least once) help the program pick a winning lottery *template* ticket.

It might be useful because the lenient approach I've implemented, which ignores any tag that doesn't belong to the in-memory XML structure representation, has a direct & definitive consequence over optional tags which pop up unaware out of the scope of the *template file*: they are just plainly ignored.

To illustrate this, imagine that you have two XML input files `myFirst.xml` and `mySecond.xml` in the same input directory, and one *optional single leaf* tag – say `Root.Row.Unexpected.Data.Tag` – appears in `mySecond.xml` but not in `myFirst.xml`. Imagine now that you run `XML2CSVGenericConverter`, and that the program chooses `myFirst.xml` as *template file*. As a result, `Root.Row.Unexpected.Data.Tag` will be completely ignored by the program which won't even acknowledge it for an acceptable *positive* or *negative filter* Xpath.

Of course, running `XML2CSVGenericConverter` N individual times (one for each XML input file) is a good workaround because when a single XML input file is processed *template* and actual data are altogether the same. But it might become somewhat tedious when you have dozens of XML input files to process.

So, because I did not want to systematically parse all XML input files just to make sure I would not miss any overdue hidden *optional* tag (a sheer waste of time) I ended up with another option, `-t`, and this paragraph.

3.4 - XML2CSV: *what next?*

Raw and standard optimizations are the very least any decent XML ⇨ CSV conversion program might do, and just what most users will ask from it.

As for the additional extensive #1 and #2 optimization flavors (respective codes: `XV1` and `XV2`, see § 4.1 -), they are just tryouts, in some way. To be honest, I'm still looking for the best conversion algorithm, simple and efficient in every respect, which would smoothly produce CSV files holding the fewest number of lines possible ([8]). When I've found it I will come back with another version holding a definitive `XV3` optimization variant (but keeping `XV1` and `XV2`, both for legacy and as a reminder of my hazardous wanderings).

Until then, I'm afraid you'll have to do with `XV1` and `XV2` and their deficiencies (that I let you discover by yourself, for it wouldn't be fun otherwise).

Regarding all the other options added I implemented them one by one while I was writing this documentation until I had no more ideas of features to add, no more paragraph to remove from § 3.4 - , and no time left either.

I'm lying, you know? I was just running out of `GetOpt` command line letters – that's all. Ho ho ho.

Anyway: some options might happen to be helpful.

Sometime.

8 *"Un truc qui dépouille tout le monde"*

4 - XML2CSV-Console-Command

`XML2CSVConsoleCommand` is the main class of the executable Jar package and makes it possible for end users (I mean, humans) to interact with the underneath core & versatile `XML2CSVGenericGenerator` class (see § 3.2 -).

The paragraph below lists all the command line parameters which are supported by `XML2CSVConsoleCommand` making it possible to tailor execution to your specific needs.

Invoking `XML2CSVConsoleCommand` with an unknown parameter, or a supported parameter but in an unsupported way will lead to hasty program termination to avoid unexpected behavior (⇨ *graceful failure*).

4.1 - Parameter list

Parameter	Description
`-h`	*Help.* This option displays the on-line help and then aborts the program immediately no matter what the other parameters might be. Optional.
`-m`	*Mute.* When this option is set nothing is displayed at runtime and the program's return code only might indicate success/failure. See § 4.2 - . Optional. Option `-m` should be placed at the head of the parameter list in order to be fully effective. *Option `-m` overrides options `-v` and `-d`.*
`-v`	*Verbose.* When this option is set extra progression messages are displayed. Optional.
`-d`{degree}	*Debug.* When this option is set debug messages are displayed. Optional. Option `-d` can be immediately followed (that is: without any gap) by an explicit degree value (`1`, `2`, or `3`) or provided alone (which transparently defaults to `1`); the higher the debug degree is, the more debug messages will be displayed at runtime. ***Warning**: debug messages are extremely time consuming and should not be activated when big XML files are processed (it could slow down computation by at least two orders of magnitude).*
`-a`	*Attributes.* Extracts element's attributes as well. Optional.
`-r`	*Raw.* Optimization deactivation. Generates CSV output file(s) in which each extracted XML tag is placed alone in its own CSV line (see § 3.3.3.1 -). Optional. *Options `-r` and `-x` are exclusive.*
`-x`{variant}	*Extensive.* Elaborate optimization level. Generates CSV output file(s) where related data are packed on the same line (see

Parameter	Description
	§ 3.3.3 -) and eventually duplicated on repeated element's lines. Optional. Option `-x` can be immediately followed (that is: without any gap) by an explicit optimization variant name (`XV1` or `XV2`) or provided alone (which transparently defaults to `XV2`). `XV1` is the command line name of extensive optimization – variant 1 (see § 3.3.3.3 -). `XV2` is the command line name of extensive optimization – variant 2 (see § 3.3.3.4 -). *Options `-x` and `-r` are exclusive.*
`-i` {input dir/file}	*Input.* Path to an XML file or path to a directory containing XML files to convert to CSV. The path might be relative or absolute. A relative path is calculated against the local Jar directory. A path containing blank characters has to be enclosed by double quotes (example: `-i "./xml files/myFile.xml"`).
`-t` {template filename}	*Template.* This option makes it possible to choose which of the input XML files (listed by the previous option `-i`) will serve for structure analysis (see § 3.3.5 -). A correct *template filename* consists of the file name only without its path. Optional. *The program picks up at random one of the input XML files when left on its own.*
`-o` {output dir}	*Output.* Path to an existing directory where CSV output file(s) will be generated. Optional. *The program uses the input directory when left on its own.*
`-b`{blend filename}	*Blend.* When this option is set the program generates a single CSV output file instead of generating as many CSV output files as XML input files (keeping names, changing extensions to CSV). Optional. Option `-b` can either be provided alone or immediately followed (that is: without any gap) by a *blend filename*. In the former case the program will generate a single CSV output file named `output.csv` (the actual file location will depend on option `-o`) and in the latter case the single CSV output file will be named after the *blend filename*.
`-e` {output encoding}	*Encoding.* This option makes it possible to choose the character encoding of the CSV output file(s). Optional. *The program defaults to `UTF-8` when left on its own.*
`-s` {output separator}	*Separator.* This option makes it possible to choose the field separator in the CSV output file(s). Optional. *The program defaults to the semicolon when left on its own.*
`-p` {+ filter file}	*Positive filtering.* Path to an `UTF-8` encoded text file (without Byte Order Mark) providing in column an explicit list of element Xpaths to extract from the XML files (while the rest will be discarded). See § 3.3.4.1 - .

Parameter	Description
	Optional. *The program extracts all XML fields when left on its own.* *Options `-p` and `-n` are exclusive.*
`-n` {- filter file}	*Negative filtering.* Path to an `UTF-8` encoded text file (without Byte Order Mark) providing in column an explicit list of element Xpaths to discard from the XML files (while the rest will be extracted). See § 3.3.4.2 - . Optional. *The program extracts all XML fields when left on its own.* *Options `-n` and `-p` are exclusive.*
`-l`{log4J config}	*Logging.* This option activates Log4J logging. Optional. Option `-l` might be immediately followed (that is: without any gap) by the path to a custom `UTF-8` encoded Log4J file (without Byte Order Mark) to use instead of the default built-in Log4J configuration file `xml2csvlog4j.properties`. xml2csvlog4j.prope rties Option `-l` provided alone activates Log4J logging using the built-in `xml2csvlog4j.properties` configuration file which creates a log file named `XML2CSV-Generic-Converter.log` in the same directory as the Jar file.
`-c`{limit}	*Cutoff.* This option activates cutoff limit, that is, a row count threshold for all CSV output files. When the threshold is active the program splits automatically output files if it is needed (additional CSV files are suffixed by `-2`, `-3`, …). Optional. Option `-c` might be immediately followed (that is: without any gap) by an integer to use instead of the default built-in `1024` limit value. The actual row count threshold is `1024x`{limit} rows per CSV file (that is, with the default limit, `1024x1024` lines max for each CSV output file).
`-w`	*Warding*. Performs *name space* aware parsing. The default behavior is to confidently ignore XML elements' *name space* aliases (that is, for instance, to read `tagName` when parsing `<lib:tagName>`). When this option is set: • the actual *name space* list is recalled atop of the CSV output (before the CSV column names, displaying something like "`NAMESPACES:` {alias1}={namespace1} {alias2}={namespace2} …"); • each element is fully qualified by its prefix, if any (namely one of the previous *name space* aliases) and the CSV column names reflect this, keeping separating colons between an element's prefix and the element's short name (example: `Root.lib1:Row.lib2:Data.lib3:Tag`). Optional. You won't need *name space* aware parsing unless your local XML contains *lookalikes* which must be sorted out (that is, two *children* tags of the same *composite* tag having the same name but different purposes and, as such, issued from different libraries – for instance `<doc:Name>` and `<customer:Name>`).

Parameter	Description
	⚠ ***Warning**: space full parsing wards off lookalike issues but obfuscates CSV column naming and complicates filter file's syntax as well (with the previous example:* `Root.lib1:Row.lib2:Data.lib3:Tag` *instead of* `Root.Row.Data.Tag` *both as a column name and as a valid filter XPath entry).*

4.2 - Return codes

You won't really care about the return codes listed below unless you decide to include an `XML2CSVGenericConverter` call in a command script (for instance: a bash Unix shell script) in connection with some kind of automated process.

It is something which concerns more advanced users; if it happens that you don't know what to do with those return codes you might as well just skip this paragraph without any regrets.

Return code	Description
0	*OK.* Termination without error.
1	*Canceled,* for instance because the help was displayed or because there was no XML file to process. *It is not an error but the program did not generate anything either.*
2	*Aborted,* because of some bad parameters (see § 4.1 -).
3	*Aborted,* because of filter file issues (see § 3.3.4 -).
4	*Aborted,* because XML structure analysis failed (see § 3.3.5 -).
5	*Aborted,* because XML data extraction failed.
6	*Aborted,* because of an unexpected exception/problem.

5 - XML2CSV-Generic-Generator

`XML2CSVGenericGenerator` is the core programmatic class one programmer would use in order to include an XML2CSV generic conversion in his/her program as an internal Java routine call.

This paragraph will be pointless for end users who are just interested in command line execution.

Programmers might find it interesting in conjunction with the Javadoc available in the `doc` directory packaged with the Jar.

The enclosing `XML2CSVConsoleCommand` class shipped as the main class of the Jar package just:

- reads a bunch of command line options thanks to an internal `GetOpt` call and then,
- controls them, and, if it happens that the parameters are correct,
- creates an appropriate `XML2CSVGenericGenerator` class instance and, finally,
- calls its `generate` method once for each XML input file.

Appropriate logging behavior (that is: mute execution, regular logging or extra verbose/debug message display) is delegated to a `LoggingFacade` abstract class which nests a few `public static` variables in relation, making it possible to control runtime logging depth from outside the generator.

5.1 - `XML2CSVGenericGenerator` constructors

5.1.1 - `File` oriented constructors

There are two file-oriented `XML2CSVGenericGenerator` instance constructors:

```
// XML2CSV output file oriented constructor.
public XML2CSVGenericGenerator(java.io.File csvOutputFile,
                               java.io.File csvOutputDir) throws XML2CSVException
```

This is a convenience constructor, equivalent to an `XML2CSVGenericGenerator(csvOutputFile, csvOutputDir, null, null, null, -1)` call.

```
// XML2CSV output file oriented constructor.
public XML2CSVGenericGenerator(java.io.File csvOutputFile,
                               java.io.File csvOutputDir,
                               java.lang.String csvFieldSeparator,
                               java.nio.charset.Charset encoding,
                               XML2CSVOptimization level,
                               long cutoff) throws XML2CSVException
```

This constructor builds an `XML2CSVGenericGenerator` instance which:

- uses its `csvOutputFile` parameter as CSV output target if left non `null`;
- uses its `csvOutputDir` parameter as CSV output directory if left non `null` (letting the program name CSV output file(s) automatically after the XML input file names).

When both `csvOutputFile` and `csvOutputDir` are provided `csvOutputFile` takes precedence.

The three next parameters `csvFieldSeparator`, `encoding` and `level` pilot CSV output generation so that:

- `csvFieldSeparator` be used as CSV field separator in the output if left non `null` instead of the default character (a semicolon);
- `encoding` be used as output charset if left non `null` instead of the default `UTF-8` character encoding;
- a custom optimization `level` be used instead of the default optimization level (standard - see § 3.3.3 -) if left non `null`.

The actual logging configuration used at runtime (mute execution, verbose or debug levels activated/deactivated, built-in console logging or Log4J logging) is inherited from the current `LoggingFacade` configuration (see § 5.4 -).

The last parameter, `cutoff`, when strictly positive, activates a built-in behavior which slices automatically any output file in parts so that no output file hold more than `cutoff` lines (adding automatically `-2`, `-3`, … suffixes to output filenames).

Just after a successful construction, a new `XML2CSVGenericGenerator` instance is ready to use (marked "*output ready*") and its `generate` method might be called immediately afterward.

5.1.2 - `OutputStream` oriented constructors

There are two stream-oriented `XML2CSVGenericGenerator` instance constructors:

```
// XML2CSV output stream oriented constructor.
public XML2CSVGenericGenerator(java.io.OutputStream output) throws XML2CSVException
```

This is a convenience constructor, equivalent to a `XML2CSVGenericGenerator(output, null, null, null)` call.

```
// XML2CSV output stream oriented constructor.
public XML2CSVGenericGenerator(java.io.OutputStream output,
                               java.lang.String csvFieldSeparator,
                               java.nio.charset.Charset encoding,
                               XML2CSVOptimization level) throws XML2CSVException
```

This constructor builds an `XML2CSVGenericGenerator` instance which uses its `output` parameter as CSV output target (`null` value forbidden).

This direct `output` is expected opened & ready, and will remain open in all circumstances (it's the caller's responsibility to close it properly at its own level).

The other parameters have the same purpose as those provided to the previous constructor (see § 5.1.1 -).

Just like before, after a successful construction, a new `XML2CSVGenericGenerator` instance is ready to use (marked "*output ready*") and its `generate` method might be called immediately afterward.

When a single output stream is used cutoff behavior is impossible, and thresholding is automatically deactivated.

5.2 - `XML2CSVGenericGenerator` generation methods

`XML2CSVGenericGenerator` defines two `generate` methods:

```
// XML2CSV conversion.
public void generate(java.io.File[] xmlInputFiles,
                               boolean withAttributes) throws XML2CSVException
```

This is a convenience method, equivalent to a `generate(xmlInputFiles, null, null, null, withAttributes)` call.

```
// XML2CSV conversion.
public void generate(java.io.File[] xmlInputFiles,
                               java.lang.String xmlTemplateFileName,
                               java.lang.String[] xmlExpectedElementsXPaths,
                               java.lang.String[] xmlDiscardedElementsXPaths,
                               boolean withAttributes) throws XML2CSVException
```

Method `generate` will raise an `XML2CSVException` if it is called while the instance is not "*output ready*".

A newly built `XML2CSVGenericGenerator` is always marked "*output ready*", and method `generate` will use the CSV output configuration provided to the constructor in the first place.

When method `generate` terminates the instance is marked "*idle*" and one of the three `setOutputStream`, `setOutputFile` or `setOutputDir` methods has to be called before the instance gets "*output ready*" again (see § 5.3 -).

A direct `output` (either provided to the constructor or by an explicit `setOutputStream` call) is expected opened & ready, and will remain open when `generate` terminates (responsibility left to the caller). On the contrary, regular files (either provided to the constructor or by an explicit `setOutputFile` or `setOutputDir` call) are automatically closed.

The XML input file(s) to process are provided by the `xmlInputFiles` parameter which cannot be left `null` or empty (an `XML2CSVException` would be raised).

The `xmlTemplateFileName` parameter, when not `null`, provides the name (without path) of the XML input file which should serve as XML *template file* (see § 3.3.5.1 -).

Parameters `xmlExpectedElementsXPaths` and `xmlDiscardedElementsXPaths` might be left `null`, but cannot be both non `null` (an `XML2CSVException` would be raised).

Parameter `xmlExpectedElementsXPaths` represents a *positive* Xpath *filter* list, that is, an explicit list of element Xpaths to extract from the XML input files, while the rest of XML elements will be discarded (see § 3.3.4.1 -).

Parameter `xmlDiscardedElementsXPaths` represents a *negative* Xpath *filter* list, that is, an explicit list of element Xpaths to discard from the XML input files, while the rest of XML elements will be fetched (see § 3.3.4.2 -).

The last parameter, `withAttributes`, must be set to `true` if element attributes have to be extracted alongside parsing, or set to `false` to have them discarded (see § 3.3.4.1 -).

5.3 - Other `XML2CSVGenericGenerator` methods

5.3.1 - Method `setOutputStream`

When called, the direct `output` parameter provided (an anonymous `OutputStream`, supposed opened & ready) becomes the new CSV output destination of the `XML2CSVGenericGenerator` instance which is is marked "*output ready*" if it wasn't ready yet.

A call to this method deactivates cutoff behavior.

5.3.2 - Method `setOutputFile`

When called, the `csvOutputFile` parameter provided (a `File`) becomes the new CSV output destination of the `XML2CSVGenericGenerator` instance which is is marked "*output ready*" if it wasn't ready yet.

5.3.3 - Method `setOutputDir`

When called, the `csvOutputDir` parameter provided (a `File`) becomes the new CSV output destination of the `XML2CSVGenericGenerator` instance which is is marked "*output ready*" if it wasn't ready yet.

5.3.4 - Method `setFieldSeparator`

Just like its name suggests, this method makes it possible to change or set the field separator that this `XML2CSVGenericGenerator` instance will use in CSV output.

5.3.5 - Method `setEncoding`

Just like its name suggests, this method makes it possible to change or set the character encoding that this `XML2CSVGenericGenerator` instance will use in CSV output.

5.3.6 - Method `setOptimization`

Just like its name suggests, this method makes it possible to change or set the optimization level that this `XML2CSVGenericGenerator` instance will use in CSV output (see § 3.3.3 -).

5.3.7 - Method `setCutoff`

Just like its name suggests, this method makes it possible to change or set the cutoff limit that this `XML2CSVGenericGenerator` instance will use in CSV output.

Cutoff behavior is deactivated when a direct `OutputStream` is used (see § 5.1.2 -).

5.3.8 - Method `setWarding`

The parameter provided (`true` or `false`) activates or inactivates *name space* aware parsing for this `XML2CSVGenericGenerator` instance.

`XML2CSVGenericGenerator` performs space less parsing with the default settings unless an explicit `setWarding(true)` call be performed before actual CSV generation.

5.4 - `LoggingFacade` abstract class

This abstract class is the internal logging facade of the program in charge of progression message display and/or error message display.

The logger's static `log` methods are transparently called by `XML2CSVGenericGenerator` and `XML2CSVConsoleCommand` each time a new message has to be displayed.

The built-in logger displays messages to the console when its `LogginFacade.log` variable is `null` (its default value), or to the underneath Log4J `org.apache.commons.logging.Log` object when set.

Apart from `LogginFacade.log`, programmers are concerned by four other `public static` variables which control the whole program logging capabilities, namely:

- `LogginFacade.MUTE_MODE`: mute execution. Deactivates message logging/display when set to `true`;
- `LogginFacade.DEBUG_MODE`: when set to `true`, activates the logging/display of debug messages during execution, tailored by the current debug degree. Overridden by mute mode;
- `LogginFacade.debugDegree`: when the logging/display of debug messages is activated, controls the debug degree, from the lowest to the highest. Correct values are `1` (the default), `2` or `3`.
- `LogginFacade.VERBOSE_MODE`: when set to `true`, activates the logging/display of verbose progression messages. Overridden by mute mode.

lochrann@rocketmail.com